



# L'Instruction Set Architecture ed il Linguaggio Assembly

Prof. Alberto Borghese  
Dipartimento di Scienze dell'Informazione  
[borgnese@dsi.unimi.it](mailto:borgnese@dsi.unimi.it)

Università degli Studi di Milano

Riferimento Patterson: Capitolo 2 (fino a 2.3), A.1, A.2, A.5



## Sommario

L'ISA ed il linguaggio macchina

L'Assembly

I registri

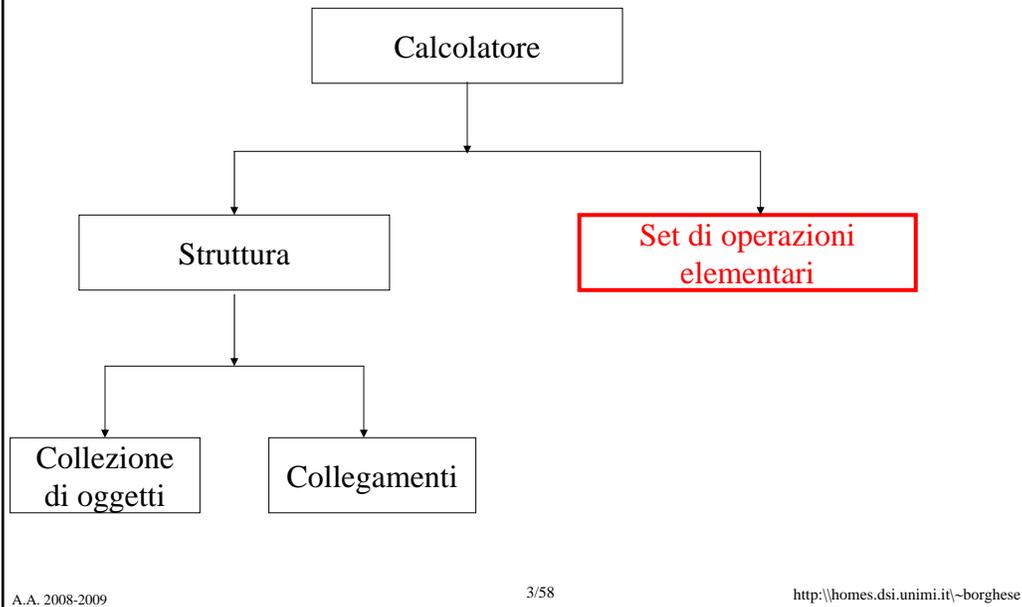
I tipi di istruzioni: istruzioni aritmetiche

Organizzazione della memoria

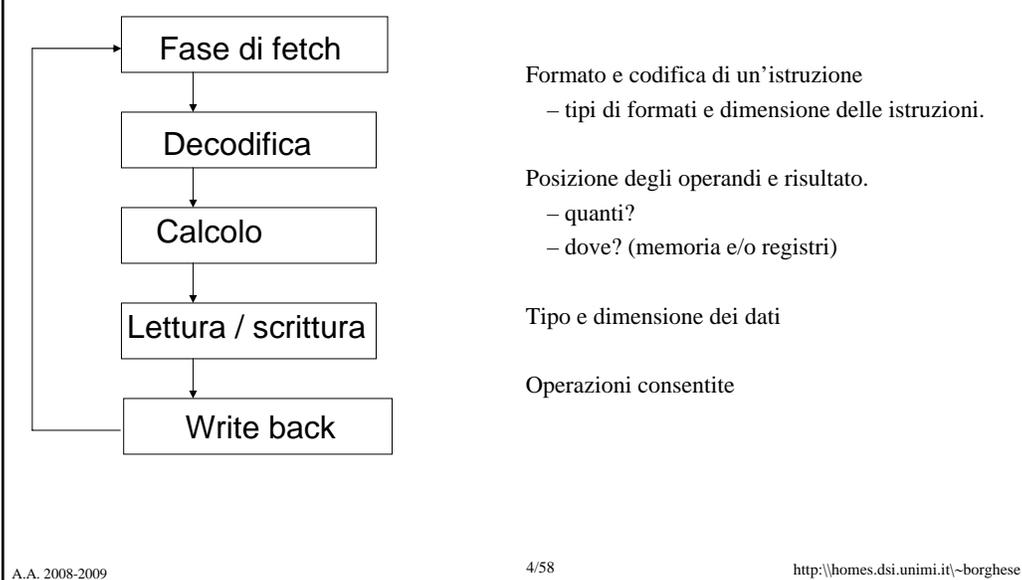
Istruzioni di accesso alla memoria.



## Descrizione di un elaboratore



## Caratteristiche di un'ISA





## Definizione di un'ISA



*Definizione del funzionamento: insieme delle istruzioni (interfaccia verso i linguaggi ad alto livello).*

- Tipologia di istruzioni.
- Meccanismo di funzionamento.

*Definizione del formato: codifica delle istruzioni (interfaccia verso l'HW).*

- Formato delle istruzioni.
- Suddivisione in gruppi omogenei dei bit che costituiscono l'istruzione.



## Tipi di istruzioni



- Le istruzioni comprese nel linguaggio macchina di ogni calcolatore possono essere classificate nelle seguenti quattro categorie:
  - Istruzioni aritmetico-logiche;
  - Istruzioni di trasferimento da/verso la memoria (*load/store*);
  - Istruzioni di salto condizionato e non condizionato per il controllo del flusso di programma;
  - Istruzioni di trasferimento in ingresso/uscita (I/O).



## Le istruzioni di un'ISA



Devono contenere tutte le informazioni necessarie ad eseguire il ciclo di esecuzione dell'istruzione: registri, comandi, ....

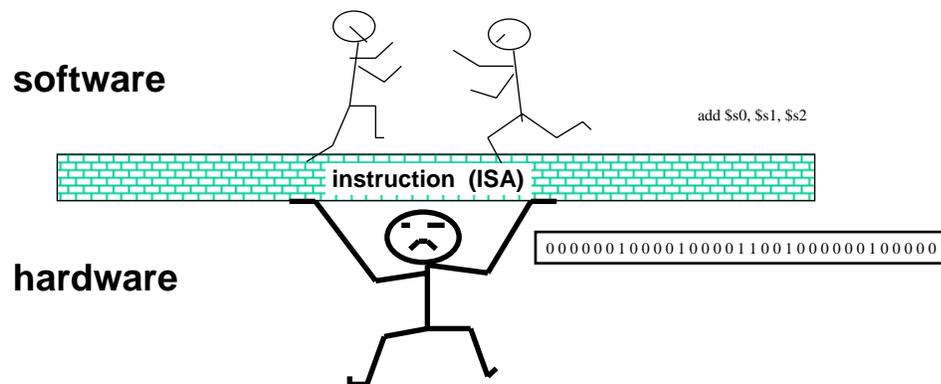
### Ogni architettura di processore ha il suo linguaggio macchina

- Architettura dell'insieme delle istruzioni elementari messe a disposizione dalla macchina (in linguaggio macchina).
  - **ISA (Instruction Set Architecture)**
- Due processori con lo stesso linguaggio macchina hanno la stessa architettura delle istruzioni anche se le implementazioni hardware possono essere diverse.
- Consente al SW di accedere direttamente all'hardware di un calcolatore.

*L'architettura delle istruzioni, specifica come vengono costruite le istruzioni in modo tale che siano comprensibili alla macchina (in linguaggio macchina).*



## Insieme delle istruzioni



**Quale è più facile modificare?**



## Sommario



L'ISA ed il linguaggio macchina

**L'Assembly**

I registri

I tipi di istruzioni: istruzioni aritmetiche

La memoria



## Le istruzioni in linguaggio macchina



- Linguaggio di programmazione direttamente comprensibile dalla macchina
  - Le parole di memoria sono interpretate come *istruzioni*
  - Vocabolario è *l'insieme delle istruzioni (instruction set)*

Programma in  
linguaggio ad alto livello (C)

```
a = a + c  
b = b + a  
var = m [a]
```



Programma in linguaggio  
macchina

```
011100010101010  
000110101000111  
000010000010000  
001000100010000
```



## Linguaggio assembly



- Le istruzioni assembly sono una rappresentazione simbolica del linguaggio macchina comprensibile dall'HW.
- Rappresentazione simbolica del linguaggio macchina
  - Più comprensibile del linguaggio macchina in quanto utilizza simboli invece che sequenze di bit
- Rispetto ai linguaggi ad alto livello, l'assembly fornisce limitate forme di controllo del flusso e non prevede articolate strutture dati
- Linguaggio usato come linguaggio target nella fase di compilazione di un programma scritto in un linguaggio ad alto livello (es: C, Pascal, ecc.)
- Vero e proprio linguaggio di programmazione che fornisce la visibilità diretta sull'hardware.



## Linguaggio C: somma dei primi 100 numeri



```
main()
{
    int i;
    int sum = 0;
    for (i = 0; i <= 100; i = i + 1)
        sum = sum + i*i;
    printf("La somma da 0 a 100 è %d\n", sum);
}
```



## Linguaggio assembly: somma dei primi 100 numeri



```
.text          addu $t9, $t8, $t4
.align 2      addu $t9, $t8, $t7
.globl main   sw $t9, 24($sp)
main:        addu $t7, $t6, 1
            subu $sp, $sp, 32
            sw $ra, 20($sp)
            sw $a0, 32($sp)
            sw $0, 24($sp)
            sw $0, 28($sp)
            .....
loop:       lw $t6, 28($sp)
            lw $t8, 24($sp)
            mult $t4, $t6, $t6
```



## Assembly come linguaggio di programmazione



- Principali *svantaggi* della programmazione in linguaggio assembly:
  - Mancanza di portabilità dei programmi su macchine diverse
  - Maggiore lunghezza, difficoltà di comprensione, facilità d'errore rispetto ai programmi scritti in un linguaggio ad alto livello
- Principali *vantaggi* della programmazione in linguaggio assembly:
  - Ottimizzazione delle prestazioni.
  - Massimo sfruttamento delle potenzialità dell'hardware sottostante.
- Le strutture di controllo hanno forme limitate
- Non esistono tipi di dati all'infuori di interi, virgola mobile e caratteri.
- La gestione delle strutture dati e delle chiamate a procedura deve essere fatta in modo esplicito dal programmatore.



## Assembly come linguaggio di programmazione



- Alcune applicazioni richiedono un approccio *ibrido* nel quale le parti più critiche del programma sono scritte in assembly (per massimizzare le prestazioni) e le altre parti sono scritte in un linguaggio ad alto livello (le prestazioni dipendono dalle capacità di ottimizzazione del compilatore).

Esempio: Sistemi embedded o dedicati

Sistemi “automatici” di traduzione da linguaggio ad alto livello (linguaggio C) ad assembly e codice binario ed implementazione circuitale (e.g. sistemi di sviluppo per FPGA).



## Sommario



L'ISA ed il linguaggio macchina

L'Assembly

**I registri**

I tipi di istruzioni: istruzioni aritmetiche

Organizzazione della memoria

Istruzioni di accesso alla memoria.



## I registri



- I registri sono associati alle variabili di un programma dal compilatore.
- Un processore possiede un numero limitato di registri: ad esempio il processore MIPS possiede **32 registri composti da 32-bit (word), register file.**
- I registri possono essere direttamente indirizzati mediante il loro numero progressivo (0, ..., 31) preceduto da \$: ad es.  
**\$0, \$1, ..., \$31**
- Per convenzione di utilizzo, sono stati introdotti nomi simbolici significativi. Sono preceduti da \$, ad esempio:  
**\$s0, \$s1, ..., \$s7 (\$s8)** Per indicare variabili in C  
**\$t0, \$t1, ... \$t9** Per indicare variabili temporanee



## Uso dei registri: convenzioni



	Nome	Numero	Utilizzo
→	\$zero	0	costante zero
	\$at	1	riservato per l'assemblatore
	\$v0-\$v1	2-3	valori di ritorno di una procedura
	\$a0-\$a3	4-7	argomenti di una procedura
→	\$t0-\$t7	8-15	registri temporanei (non salvati)
→	\$s0-\$s7	16-23	registri salvati
→	\$t8-\$t9	24-25	registri temporanei (non salvati)
	\$k0-\$k1	26-27	gestione delle eccezioni
	\$gp	28	puntatore alla global area (dati)
	\$sp	29	stack pointer
	\$s8	30	registro salvato (fp)
	\$ra	31	indirizzo di ritorno



## I registri per le operazioni floating point



- Esistono 32 registri utilizzati per l'esecuzione delle istruzioni.
- Esistono **32** registri per le operazioni floating point (virgola mobile) indicati come

**\$f0, ..., \$f31**

- Per le operazioni in doppia precisione si usano i registri contigui

**\$f0, \$f2, \$f4, ...**



## Sommario



L'ISA ed il linguaggio macchina

L'Assembly

I registri

**I tipi di istruzioni: istruzioni aritmetiche**

Organizzazione della memoria

Istruzioni di accesso alla memoria.



## Istruzioni aritmetico-logiche



- In MIPS, un'istruzione aritmetico-logica possiede in generale *tre* operandi: i due registri contenenti i valori da elaborare (*registri sorgente*) e il registro contenente il risultato (*registro destinazione*).
- L'ordine degli operandi è **fisso**: prima il registro contenente il risultato dell'operazione e poi i due operandi.
- L'istruzione assembly contiene il codice operativo e tre campi relativi ai tre operandi:

```
OPCODE  DEST,  SORG1,  SORG2
```

Le operazioni vengono eseguite esclusivamente su dati presenti nella CPU, non su dati residenti nella memoria.



## Esempi: istruzioni add e sub



Codice C:

```
R = A + B;
```

*(Diagram showing arrows from 'A' and 'B' in the C code pointing to '\$s1' and '\$s2' in the MIPS code below)*

Codice assembler MIPS:

```
add $s0, $s1, $s2  
add rd,  rs,  rt
```

mette la somma del contenuto di rs e rt in rd:

```
add rd, rs, rt    # rd ← rs + rt
```

Nella traduzione da linguaggio ad alto livello a linguaggio assembly, le variabili sono associate ai registri dal compilatore

**sub** serve per sottrarre il contenuto di due registri sorgente rs e rt:

```
sub rd rs rt
```

e mettere la differenza del contenuto di rs e rt in rd

```
sub rd, rs, rt    # rd ← rs - rt
```



## Istruzioni aritmetico-logiche in sequenza



Il fatto che ogni istruzione aritmetica ha tre operandi sempre nella stessa posizione consente di semplificare l'hw, ma complica alcune cose...

Codice C:  $Z = A - (B + C + D) \Rightarrow$   
 $E = B + C + D; Z = A - E;$

Suppongo che le variabili siano contenute nei seguenti registri:

A -> \$s0   B -> \$s1   C -> \$s2   D -> \$s3   Z -> \$s5

Codice MIPS: `add $t0, $s1, $s2`  
`add $t1, $t0, $s3`  
`sub $s5, $s0, $t1`



## Istruzioni aritmetico-logiche



- Operazioni con un numero di operandi maggiore di tre possono essere effettuate scomponendole in operazioni più semplici.
- Ad esempio, per eseguire la somma e sottrazione delle variabili A . . D nella variabile Z servono tre istruzioni :

Codice C:  $Z = A + B - C + D$

Codice MIPS: `add $t0, $s0, $s1`  
`sub $t1, $t0, $s2`  
`add $s5, $t1, $s3`



## Implementazione alternativa



- Operazioni con un numero di operandi maggiore di tre possono essere effettuate scomponendole in operazioni più semplici.
- Ad esempio, per eseguire la somma e sottrazione delle variabili A . . D nella variabile Z servono tre istruzioni :

Codice C:  $Z = (A + B) - (C - D)$

Codice MIPS:  
`add $t0, $s0, $s1`  
`sub $t1, $s2, $s3`  
`sub $s5, $t0, $t1`

Quale implementazione è la migliore? Sceglierà il compilatore il quale cerca di massimizzare la parallelizzazione del codice.



## add: varianti



- `addi $s1, $s2, 100` #add immediate
  - Somma una costante: il valore del secondo operando è presente nell'istruzione come costante e sommata estesa in segno.  
 $rt \leftarrow rs + \text{costante}$
- `addu $s0, $s1, $s2` #add unsigned
  - Evita overflow: la somma viene eseguita considerando gli addendi sempre positivi. Il bit più significativo è parte del numero e non è bit di segno.
- `addiu $s0, $s1, 100` #add immediate unsigned
  - Somma una costante ed evita overflow.



## Moltiplicazione



- Due istruzioni:
  - `mult rs rt`
  - `multu rs rt`                    `# unsigned`
- Il registro destinazione è *implicito*.
- Il risultato della moltiplicazione viene posto sempre in due registri dedicati di una parola (special purpose) denominati *hi* (*High order word*) e *lo* (*Low order word*)
- La moltiplicazione di due numeri rappresentabili con 32 bit può dare come risultato un numero non rappresentabile in 32 bit



## Moltiplicazione



- Il risultato della moltiplicazione si preleva dal registro **hi** e dal registro **lo** utilizzando le due istruzioni:

- `mfhi rd`                    `# move from hi`
  - Sposta il contenuto del registro **hi** nel registro **rd**
- `mflo rd`                    `# move from lo`
  - Sposta il contenuto del registro **lo** nel registro **rd**

Test sull'overflow

Risultato del prodotto



## Pseudoistruzioni



- Per semplificare la programmazione, MIPS fornisce un insieme di *pseudoistruzioni*
- Le pseudoistruzioni sono un modo compatto ed intuitivo di specificare un insieme di istruzioni
  - Non hanno un corrispondente 1 a 1 con le istruzioni dell'ISA.
- La traduzione della pseudoistruzione nelle istruzioni equivalenti è attuata automaticamente dall'assemblatore

### Esempi:

- **move \$t0, \$t1**
  - **add \$t0, \$zero, \$t1** # (in alternativa) **addi \$t0, \$t1, 0**
- **mul \$s0, \$t1, \$t2**
  - **mult \$t1, \$t2**
  - **mflo \$s0**
- **div \$s0, \$t1, \$t2**
  - **div \$t1, \$t2**
  - **mflo \$s0**



## Sommario



L'ISA ed il linguaggio macchina

L'Assembly

I registri

I tipi di istruzioni: istruzioni aritmetiche

**Organizzazione della memoria**

Istruzioni di accesso alla memoria.



# Memoria Principale



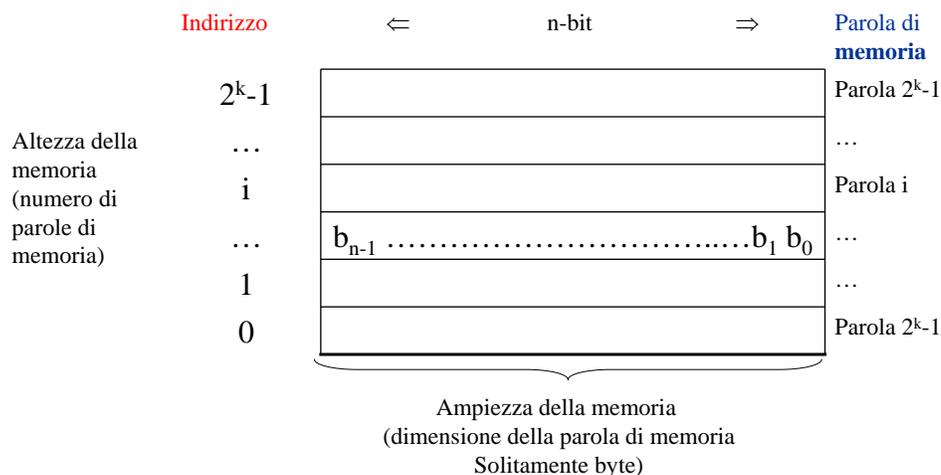
- Le memorie in cui ogni locazione può essere raggiunta in un breve e prefissato intervallo di tempo misurato a partire dall'istante in cui si specifica l'indirizzo desiderato, vengono chiamate memorie ad accesso casuale (*Random Access Memory* – *RAM*)
- Nelle RAM il *tempo di accesso alla memoria* (tempo necessario per accedere ad una parola di memoria) è *fisso e indipendente* dalla posizione della parola alla quale si vuole accedere.
- Il contenuto delle locazioni di memoria può rappresentare sia istruzioni che dati, sui quali l'architettura sta lavorando.
- La memoria può essere visto come un array monodimensionale.



# La memoria



- La memoria è vista come un unico grande array uni-dimensionale.
- Un **indirizzo di memoria** costituisce un **indice** all'interno dell'array.





## Indirizzi nella memoria principale



- La memoria è organizzata in *parole* composte da  $n$ -bit che possono essere indirizzati separatamente.
- Ogni **parola** di memoria è associata ad un **indirizzo** composto da  $k$ -bit.
- I  $2^k$  indirizzi costituiscono lo *spazio di indirizzamento* del calcolatore. Ad esempio un indirizzo composto da  $32$ -bit genera uno spazio di indirizzamento di  $2^{32}$  o  $4\text{Gbyte}$ .



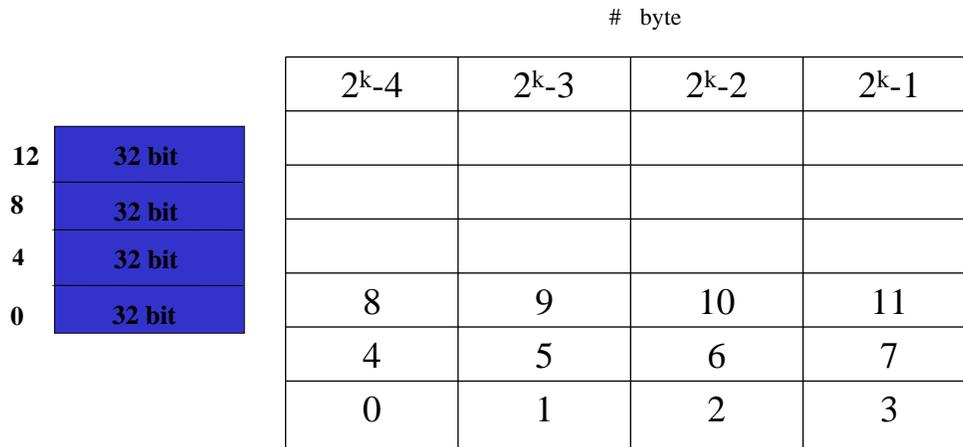
## Memoria Principale e parole



- In genere, la dimensione della parola di memoria non coincide con la dimensione dei registri contenuti nella *CPU*.
  - Per ottimizzare i tempi, ad ogni trasferimento vengono trasferiti contemporaneamente un numero di byte pari o multiplo del numero di byte che costituisce la parola dell'architettura.
    - ⇒ l'operazione di *load/store* di una parola avviene in un singolo ciclo di clock del bus.
  - Le parole hanno quindi generalmente indirizzo in memoria che è multiplo di 4.
- ⇒ Problema dell'allineamento dei dati.



## Indirizzamento della memoria MIPS



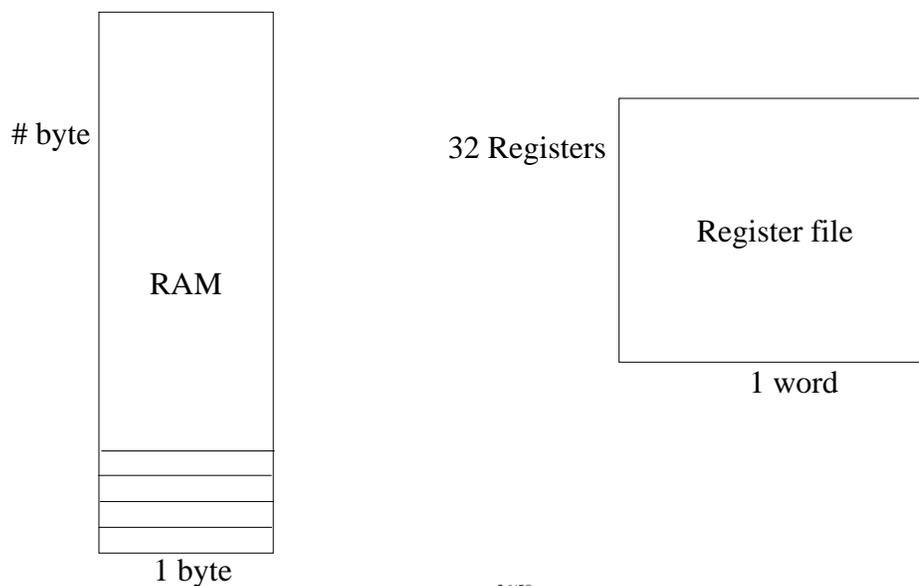
A.A. 2008-2009

35/58

<http://homes.dsi.unimi.it/~borghese>



## Memoria e Register file



A.A. 2008-2009

36/58

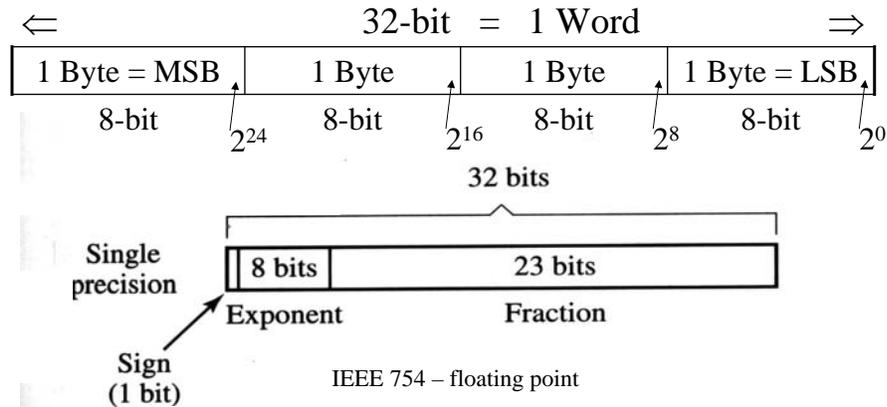
<http://homes.dsi.unimi.it/~borghese>



## Indirizzamento dei byte all'interno della parola



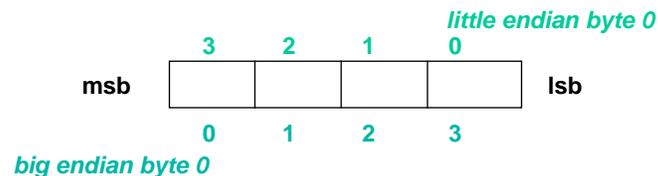
MIPS utilizza un **indirizzamento al byte**, cioè l'indice punta ad un byte di memoria, byte consecutivi hanno indirizzi consecutivi indirizzi di parole consecutive (adiacenti) differiscono di un fattore 4 (8-bit x 4 = 32-bit); ad ogni indirizzo è associato un byte.

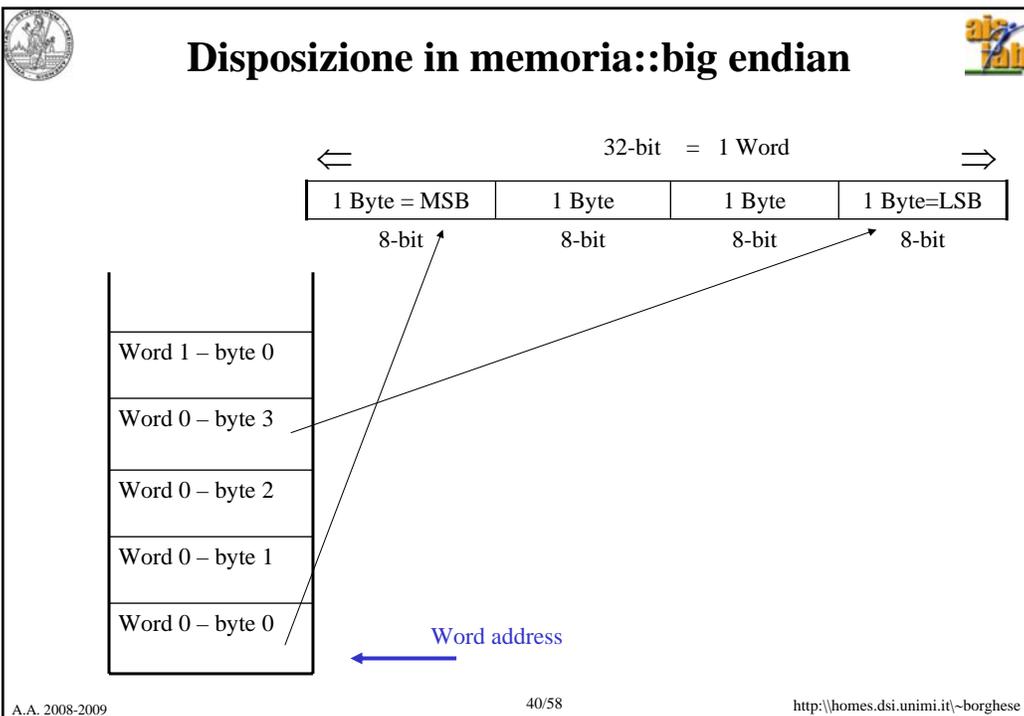
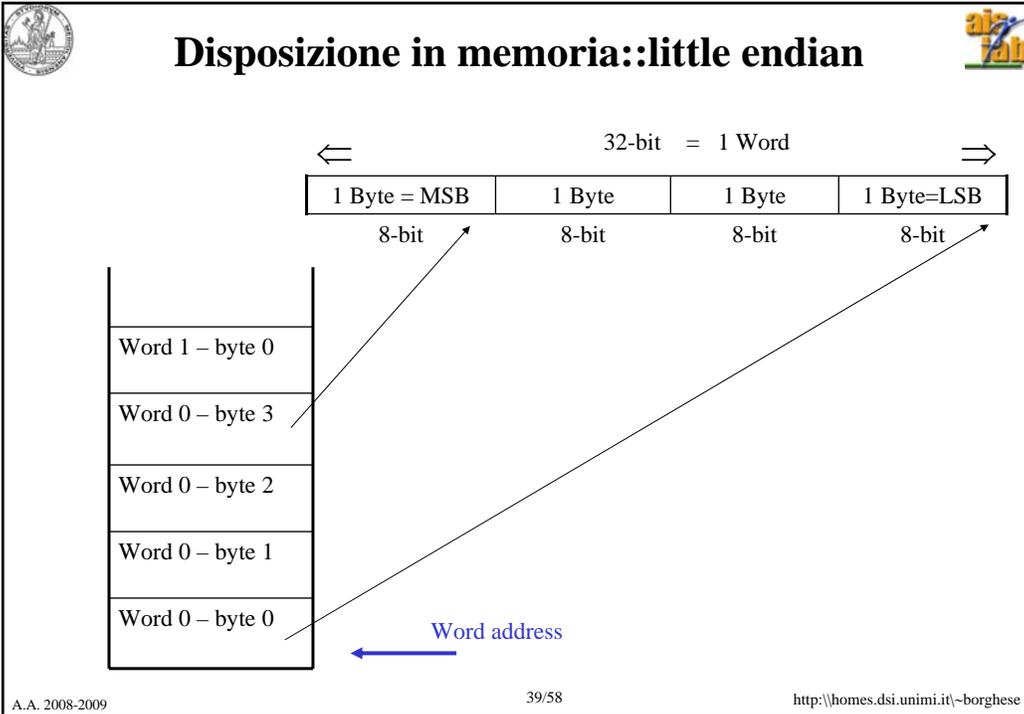


## Addressing Objects: Endianess



- **Big Endian:** address of most significant byte = word address (xx00 = Big End of word)
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP
- **Little Endian:** address of least significant byte = word address (xx00 = Little End of word)
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)







## Organizzazione logica della memoria

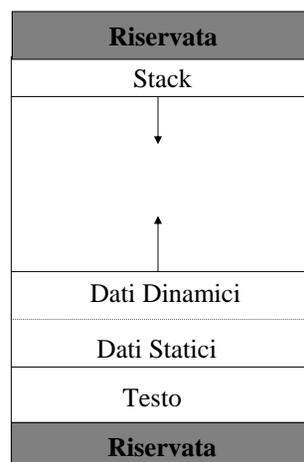


Nei sistemi basati su processore MIPS (e Intel) la memoria è solitamente divisa in **tre** parti:

- **Segmento testo:** contiene le **istruzioni** del programma
- **Segmento dati:** ulteriormente suddiviso in:
  - **dati statici:** contiene dati la cui dimensione è conosciuta al momento della compilazione e il cui intervallo di vita coincide con l'esecuzione del programma
  - **dati dinamici:** contiene dati ai quali lo spazio è allocato dinamicamente al momento dell'esecuzione del programma su richiesta del programma stesso.
- **Segmento stack:** contiene lo stack allocato automaticamente da un programma durante l'esecuzione.

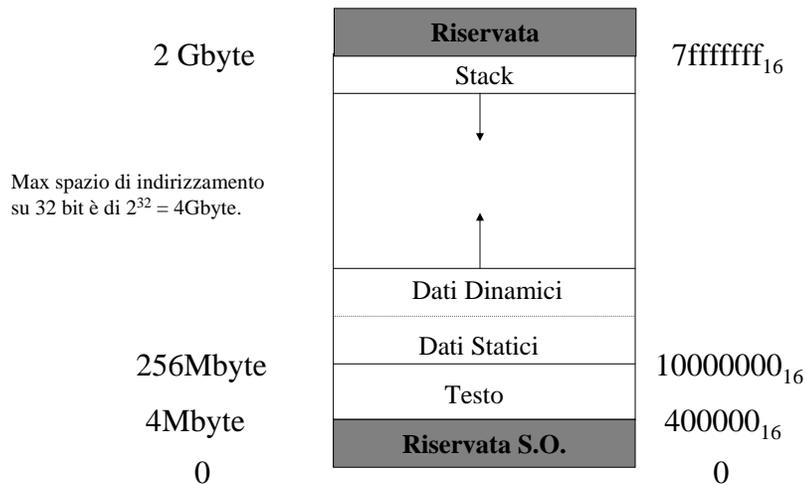


## Organizzazione logica della memoria





## Organizzazione logica della memoria



## Sommario



L'ISA ed il linguaggio macchina

L'Assembly

I registri

I tipi di istruzioni: istruzioni aritmetiche

Organizzazione della memoria

**Istruzioni di accesso alla memoria.**



## Istruzioni di trasferimento dati



- Gli operandi di una istruzione aritmetica devono risiedere nei registri che sono in numero limitato (32 nel MIPS). I programmi in genere richiedono un numero maggiore di variabili.
- Cosa succede ai programmi i cui dati richiedono più di 32 registri (32 variabili)?  
Alcuni dati risiedono in memoria.
- La tecnica di mettere le variabili meno usate (o usate successivamente) in memoria viene chiamata **Register Spilling**.



*Servono istruzioni apposite per trasferire dati da memoria a registri e viceversa*



## Istruzioni di trasferimento dati



- MIPS fornisce due operazioni base per il trasferimento dei dati:
  - **lw (load word)** per trasferire una parola di memoria in un registro della CPU
  - **sw (store word)** per trasferire il contenuto di un registro della CPU in una parola di memoria

*lw e sw richiedono come argomento l'indirizzo della locazione di memoria sulla quale devono operare*



## Istruzione *load*



- L'istruzione di *load* trasferisce una copia dei dati/istruzioni contenuti in una specifica locazione di memoria ai registri della *CPU*, lasciando inalterata la parola di memoria:

`load LOC, r1           # r1 ← [LOC]`

- La *CPU* invia l'indirizzo della locazione desiderata alla memoria e richiede un'operazione di lettura del suo contenuto.
- La memoria effettua la lettura dei dati memorizzati all'indirizzo specificato e li invia alla *CPU*.



## Istruzione di *store*



- L'istruzione di *store* trasferisce una parola di informazione dai registri della *CPU* in una specifica locazione di memoria, sovrascrivendo il contenuto precedente di quella locazione:

`store r2, LOC           # [LOC] ← r2`

- La *CPU* invia l'indirizzo della locazione desiderata alla memoria, assieme con i dati che vi devono essere scritti e richiede un'operazione di scrittura.
- La memoria effettua la scrittura dei dati all'indirizzo specificato.

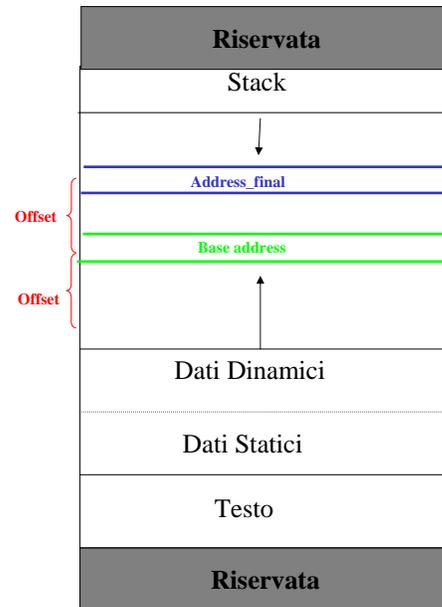


## Indirizzamento della memoria



Base + spiazamento  
Base + Offset

$$\text{Address\_final} = \text{Base\_address} + \text{Offset}$$



## Istruzione lw



- Nel MIPS, l'istruzione **lw** ha tre argomenti:
  - il *registro destinazione* in cui caricare la parola letta dalla memoria
  - una costante o *spiazamento (offset)*
  - un registro base (*base register*) che contiene il valore dell'indirizzo base (*base address*) da sommare alla costante.
- L'indirizzo della parola di memoria da caricare nel registro destinazione è ottenuto dalla somma della costante e del contenuto del registro base.



## Istruzione lw: trasferimento da memoria a registro



```
lw $s1, 100($s2)    # $s1 ← M[ [$s2] + 100 ]
```



*Al registro destinazione \$s1 è assegnato il valore contenuto all'indirizzo di memoria (\$s2 + 100) in byte.*



## Istruzione sw: trasferimento da registro a memoria



Possiede argomenti analoghi alla lw

### Esempio:

```
sw $s1, 100($s2)    # M[ [$s2] + 100 ] ← $s1
```

*Alla locazione di memoria di indirizzo (\$s2 + 100) è assegnato il valore contenuto nel registro \$s1*



## lw & sw: esempio di compilazione



Codice C:  $A[12] = h + A[8];$

- Si suppone che:
  - la variabile **h** sia associata al registro **\$s2**
  - l'indirizzo del primo elemento dell'array (*base address*) sia contenuto nel registro **\$s3** (**A[0]**)

Codice MIPS:

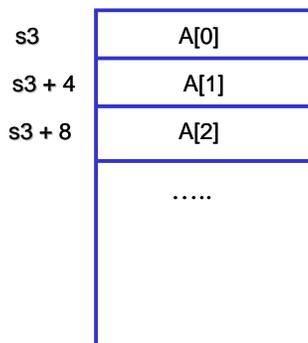
```
lw $t0, 32($s3)           # $t0 ← M[ [$s3] + 32]
add $t0, $s2, $t0         # $t0 ← $s2 + $t0
sw $t0, 48($s3)           # M[ [$s3] + 48] ← $t0
```



## Memorizzazione di un vettore



- L'elemento numero **i-esimo** di un array si troverà nella locazione  $br + 4 * i$  dove:
  - **br** è il registro base;
  - **i** è l'indice ad alto livello;
  - il fattore **4** dipende dall'indirizzamento al byte della memoria nel MIPS



A[0]	0	1	2	3
	4	5	6	7
Offset (A[2])	8	9	10	11
	$2^k-4$	$2^k-3$	$2^k-2$	$2^k-1$



## Array: esempio di lettura



- Sia A un array di N word. Realizziamo l'istruzione C:  $g = h + A[i]$
- Si suppone che:
  - le variabili **g**, **h**, **i** siano associate rispettivamente ai registri **\$s1**, **\$s2**, ed **\$s4**
  - l'indirizzo del primo elemento dell'array (*base address*) sia contenuto nel registro **\$s3**
- L'elemento **i-esimo** dell'array si trova nella locazione di memoria di indirizzo **( $\$s3 + 4 * i$ )**.
- Caricamento dell'indirizzo di A[i] nel registro temporaneo **\$t1**:

```
multi $t1, $s4, 4      # $t1 ← 4 * i
add $t1, $t1, $s3      # $t1 ← add. of A[i]
                       # that is ($s3 + 4 * i)
```
- Per trasferire A[i] nel registro temporaneo **\$t0**:

```
lw $t0, 0($t1)        # $t0 ← A[i]
```
- Per sommare h e A[i] e mettere il risultato in g:

```
add $s1, $s2, $t0     # g = h + A[i]
```

A.A. 2008-2009

55/58

<http://homes.dsi.unimi.it/~borghese>



## Array: aritmetica dei puntatori



```
for (i=0; i<N; i+=2)  -l'indirizzo del primo elemento dell'array (base
    g = h + A[i];      address) sia contenuto nel registro $s3
```

First iterations:

```
lw $t0, 0($s3)
```

All the other iterations:

```
addi $s3, $s3, 8
```

```
lw $t0, 0($s3)
```

- Increment of the address of the location of A[i], inside \$s3, by adding the proper offset.

A.A. 2008-2009

56/58

<http://homes.dsi.unimi.it/~borghese>



## Istruzioni aritmetiche vs. load/store



- Le istruzioni aritmetiche leggono il contenuto di due registri (operandi) , eseguono una computazione e scrivono il risultato in un terzo registro (destinazione o risultato)
- Le operazioni di trasferimento dati leggono e scrivono un solo operando senza effettuare nessuna computazione



## Sommario



- L'ISA ed il linguaggio macchina
- L'Assembly
- I registri
- I tipi di istruzioni: istruzioni aritmetiche
- Organizzazione della memoria
- Istruzioni di accesso alla memoria.